**BIG DATA MANAGEMENT**

**POST GRADUATE DIPLOMA**

**IN DATA ENGINEERING**

## ASSIGNMENT - 6

**SUBMITTED BY:**

**NIRAJ BHAGCHANDANI [G23AI2087]**

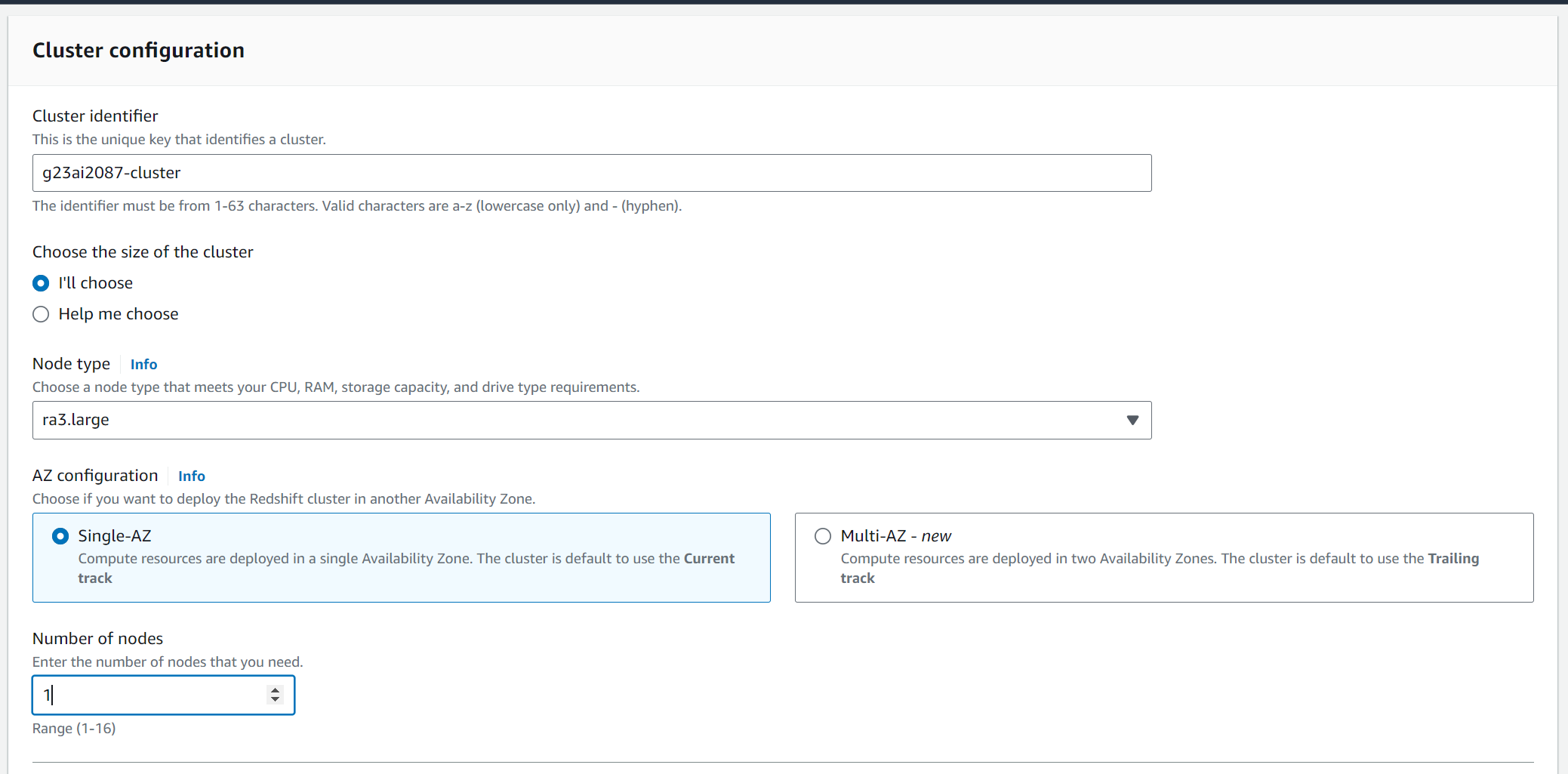
****

**SUBMISSION DATE: 15th December, 2024**

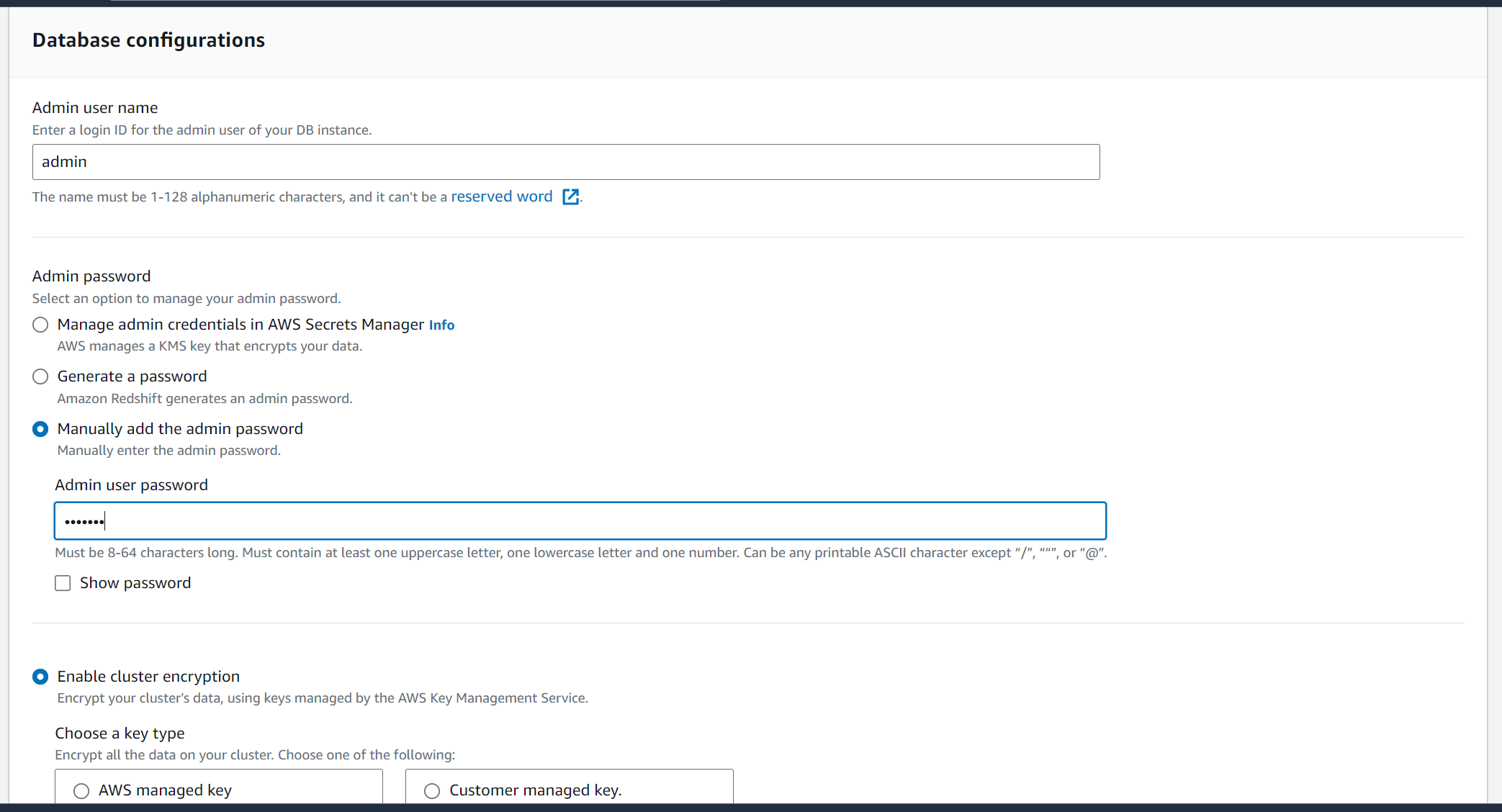
**DEPARTMENT OF AIDE**

**INDIAN INSTITUTE OF TECHNOLOGY, JODHPUR**

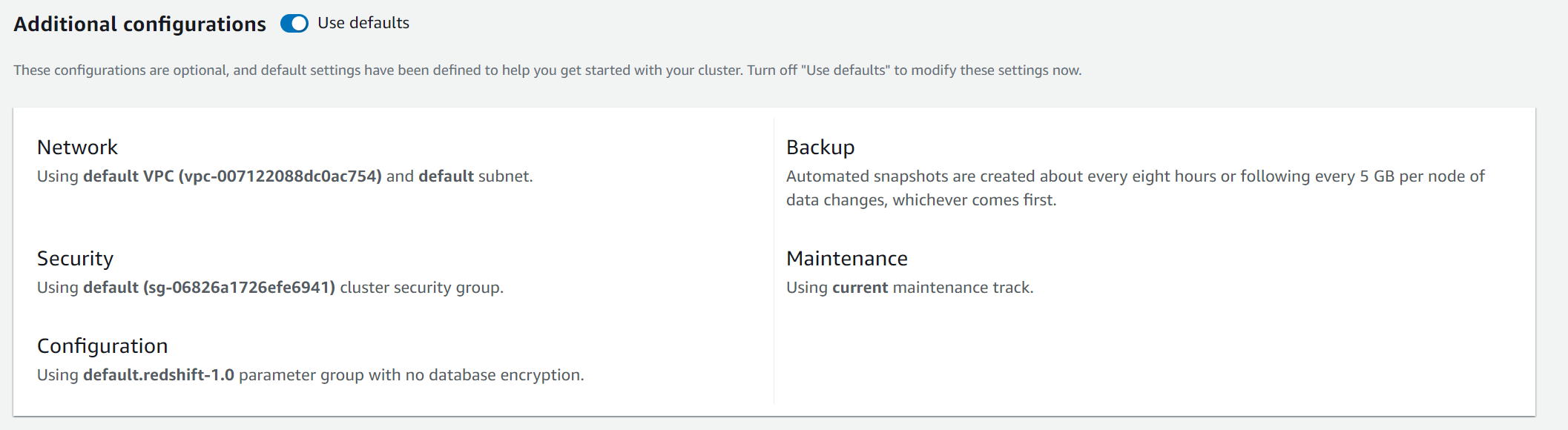
Step – 1: Set the Cluster Identifier, choose Node Type (ra3.large), select Single-AZ, and specify the Number of Nodes (1).



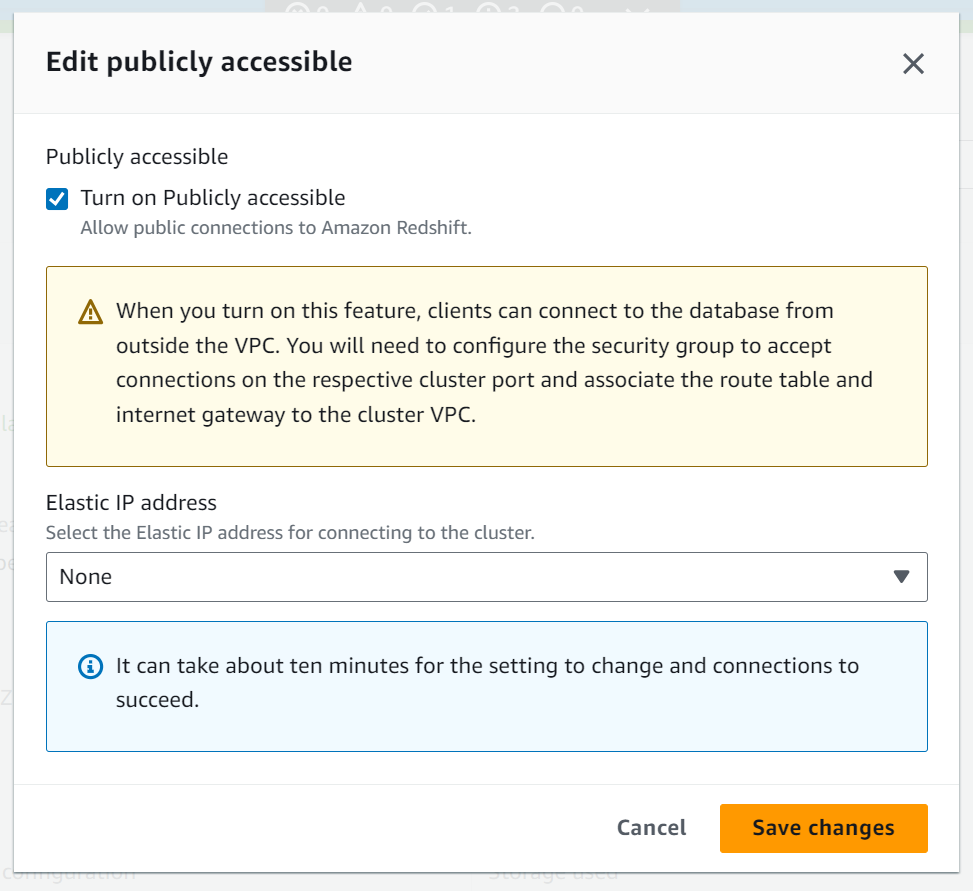
Step – 2: Set the Admin User Name (e.g., "admin"), manually add the password, and optionally enable cluster encryption with AWS or Customer managed keys.



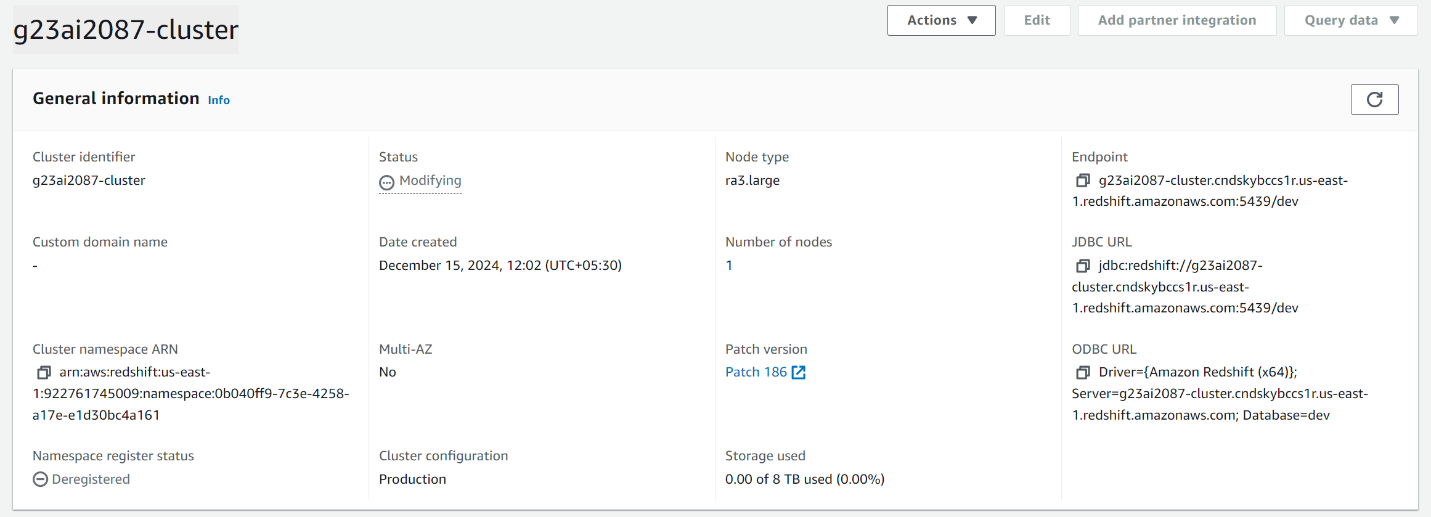
Step -3: Use default settings for Network, Security, Configuration, Backup, and Maintenance for the cluster.



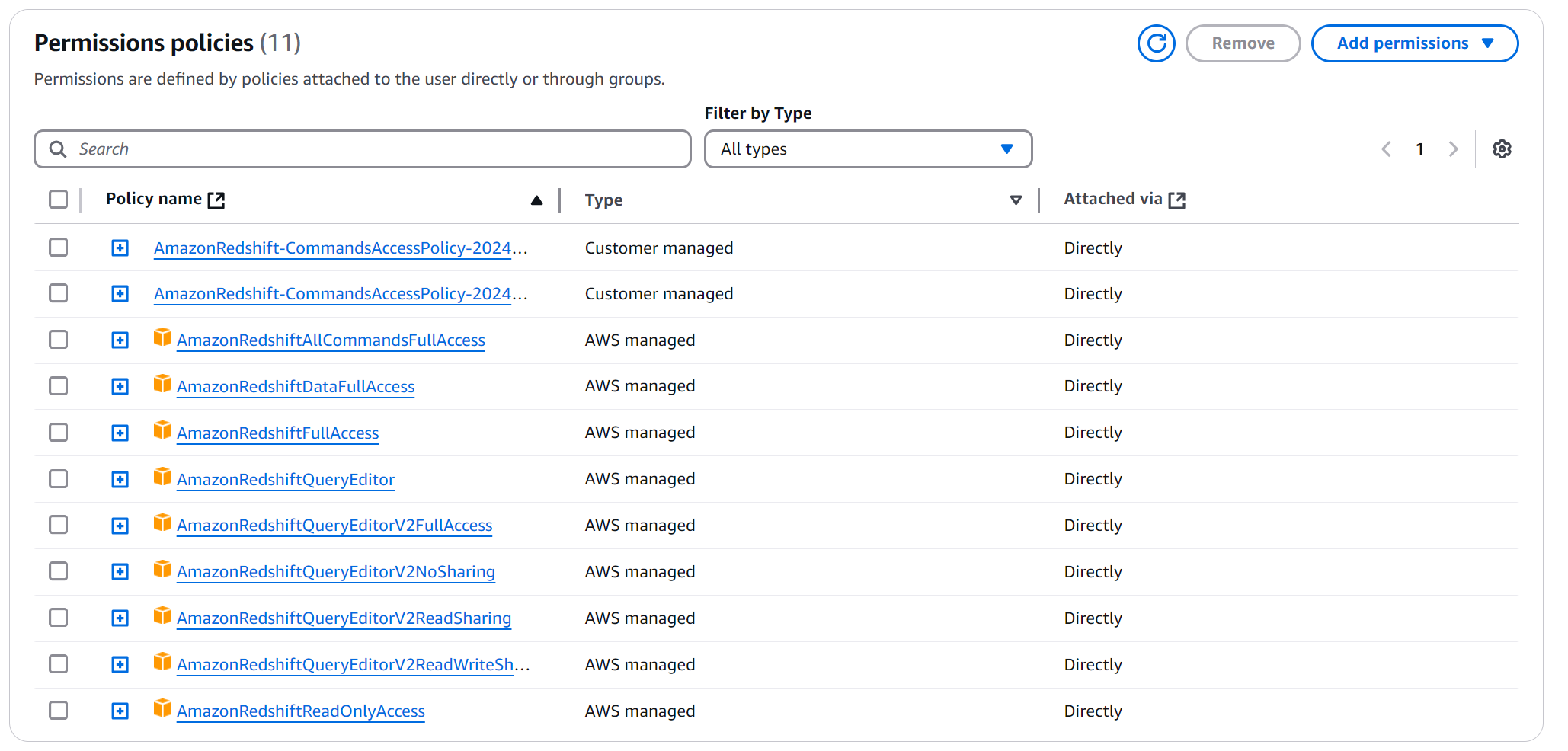
Step – 4: Enable Publicly accessible to allow public connections, configure the Elastic IP address if needed, and note the changes may take about ten minutes to apply.



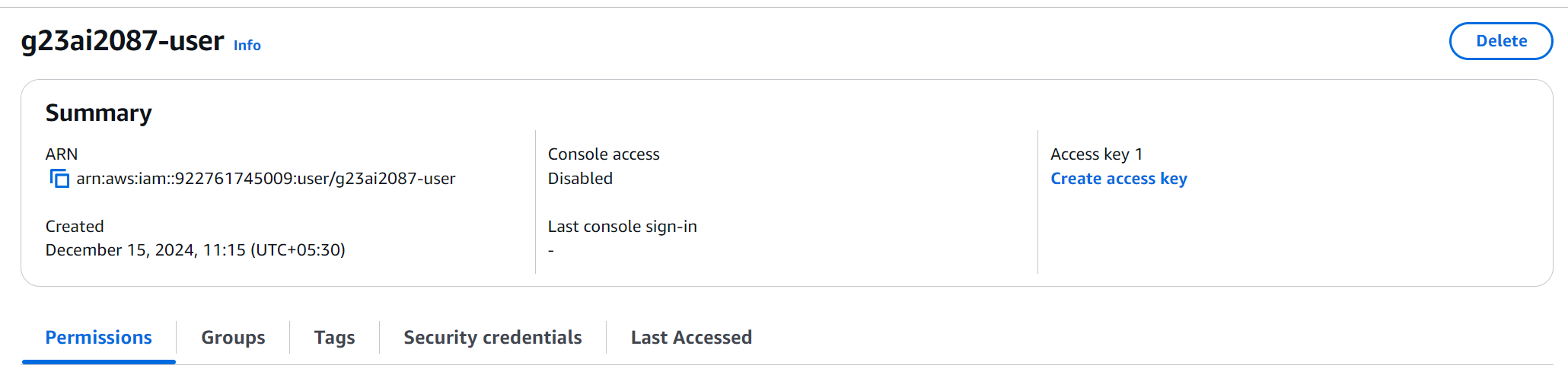
Step – 5: The g23ai2087-cluster is in a Modifying state with a Node type of ra3.large, 1 node, and is configured for Production. The cluster endpoint, JDBC URL, and ODBC URL are provided for connecting to the database.



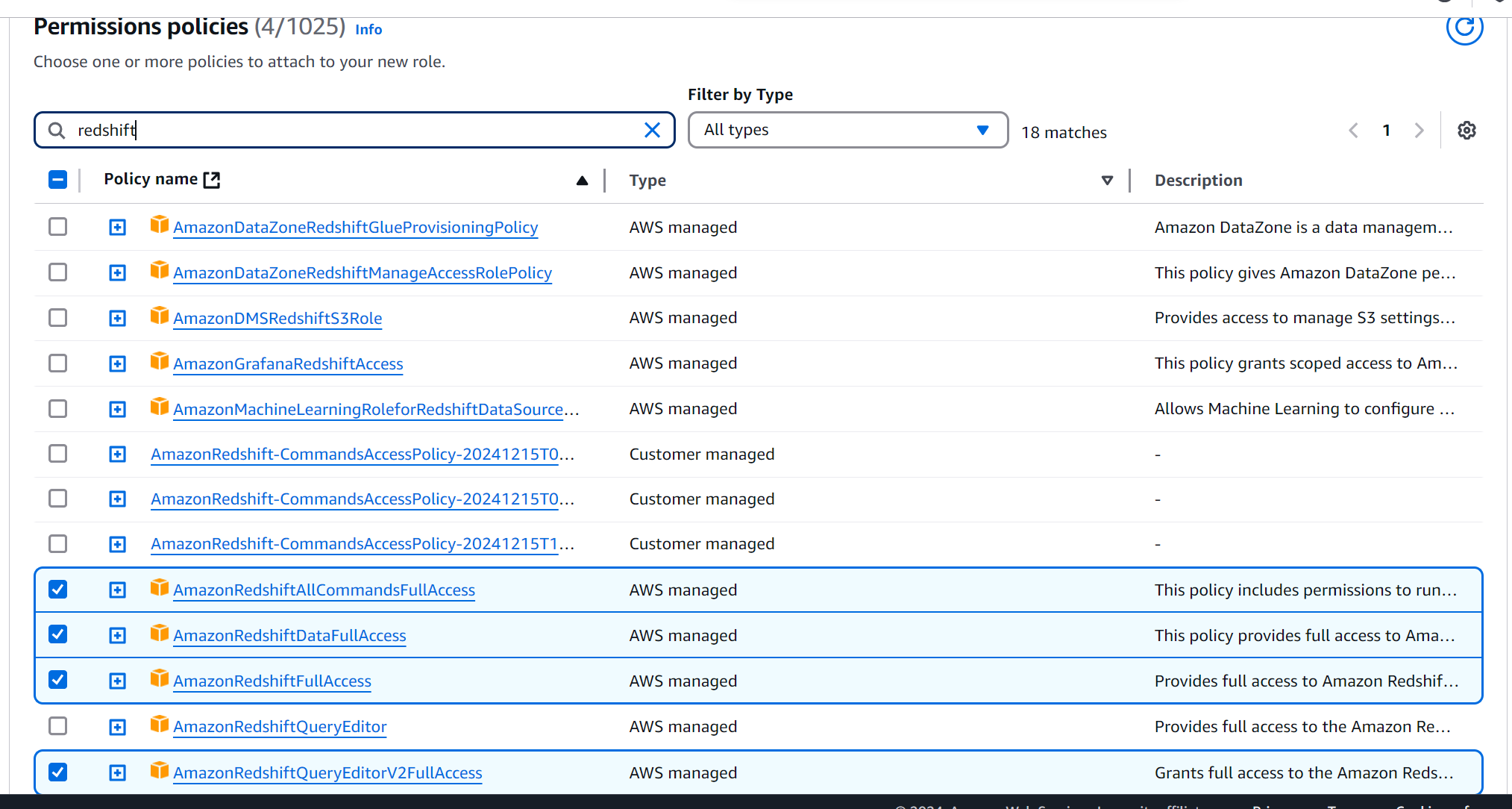
Step – 6: The Permissions policies list contains 11 policies attached directly to the user, including AWS managed policies like AmazonRedshiftFullAccess, AmazonRedshiftQueryEditorV2FullAccess, and AmazonRedshiftReadOnlyAccess, alongside Customer managed policies.



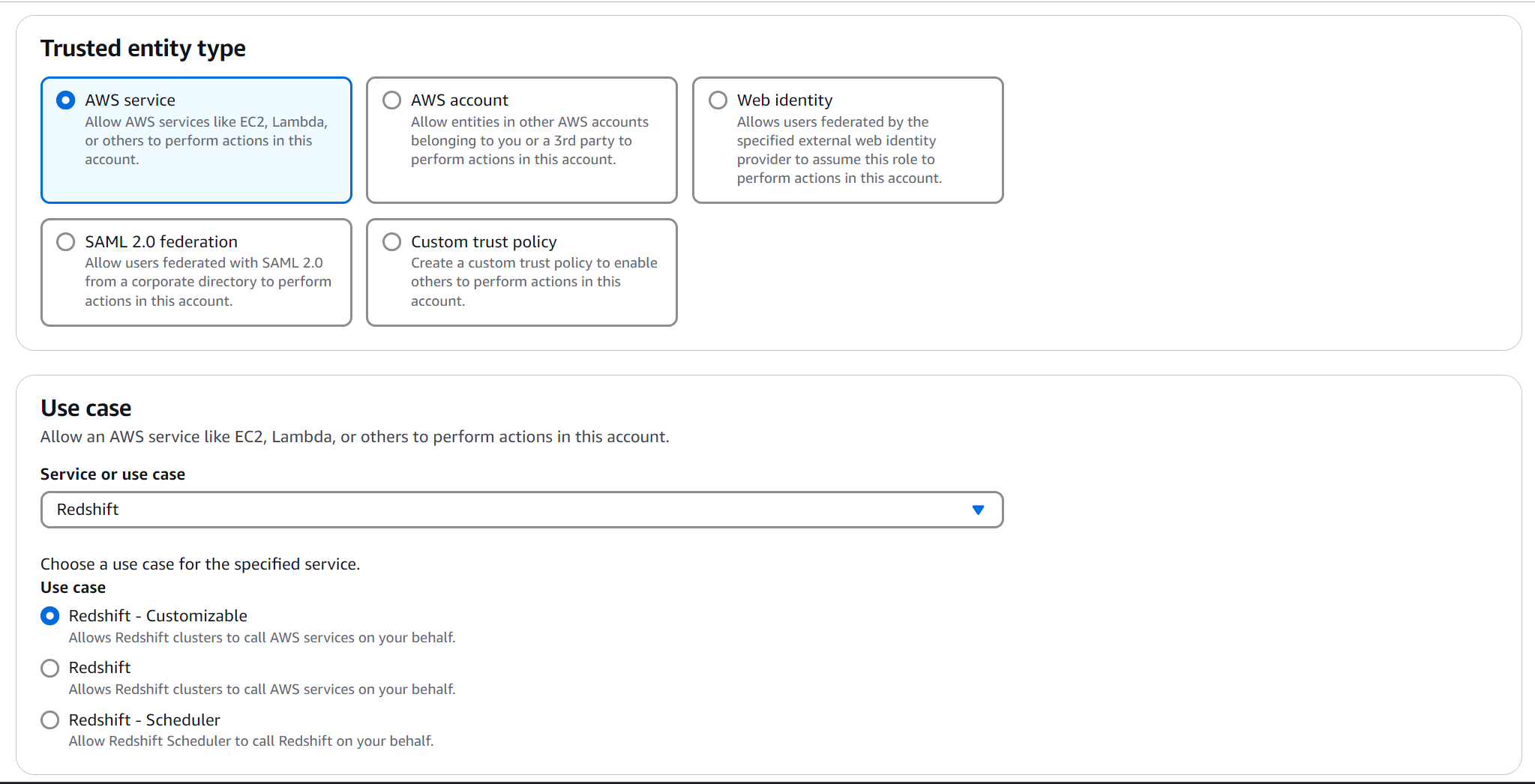
Step – 7: The g23ai2087-user has console access disabled and no recorded last console sign-in. An Access key can be created for programmatic access. The ARN and creation date (December 15, 2024) are also provided.



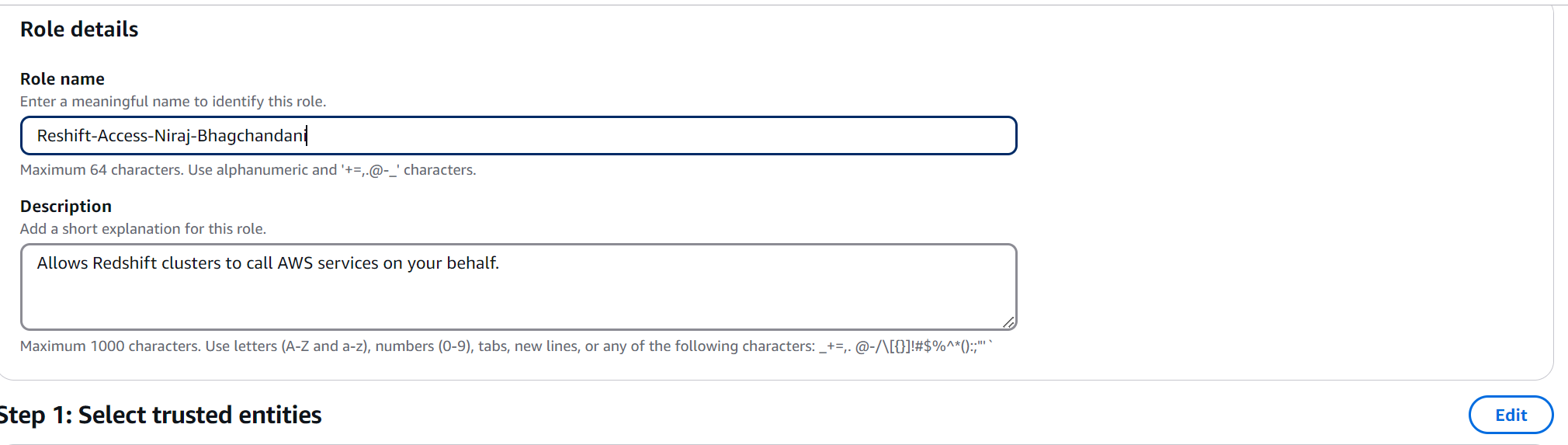
Step – 8: The Permissions policies list displays 18 matches for "Redshift", including AWS managed policies like AmazonRedshiftAllCommandsFullAccess, AmazonRedshiftDataFullAccess, and AmazonRedshiftFullAccess, with some already selected for the role.



Step – 9: The Trusted entity type is set to AWS service, with the Service selected as Redshift. The chosen Use case is Redshift - Customizable, allowing Redshift clusters to call AWS services on your behalf.



Step – 10: The Role name is set to Redshift-Access-Niraj-Bhagchandani, with a Description that states, *"Allows Redshift clusters to call AWS services on your behalf."*



Now do the following tasks with the help of the starter code provided below

1. Write the method connect() to make a connection to the database. [5]

Code:

1. public Connection connect() {
2. try {
3. Class.forName("com.amazon.redshift.jdbc42.Driver");
4. Properties properties = new Properties();
5. properties.setProperty("user", masterUsername);
6. properties.setProperty("password", password);
7. this.con = DriverManager.getConnection(redshiftUrl, properties);
8. System.out.println("Connection to Redshift established successfully.");
9. } catch (ClassNotFoundException e) {
10. System.out.println("Error: Redshift JDBC driver not found.");
11. e.printStackTrace();
12. } catch (SQLException e) {
13. System.out.println("Failed to connect to the database.");
14. e.printStackTrace();
15. }
16. return con;
17. }

Output:

A screen shot of a computer

Description automatically generated

***Fig. 6.1*** *Role configuration details for Redshift access, specifying the role name and description for AWS service permissions.*

1. Method close() to close the connection to the database. [5]

Code:

1. public void close() {
2. System.out.println("Closing database connection.");
3. try {
4. if (con != null && !con.isClosed()) {
5. con.close();
6. }
7. } catch (SQLException e) {
8. System.out.println("Error closing the connection.");
9. }
10. }

Output:

A black screen with white text

Description automatically generated

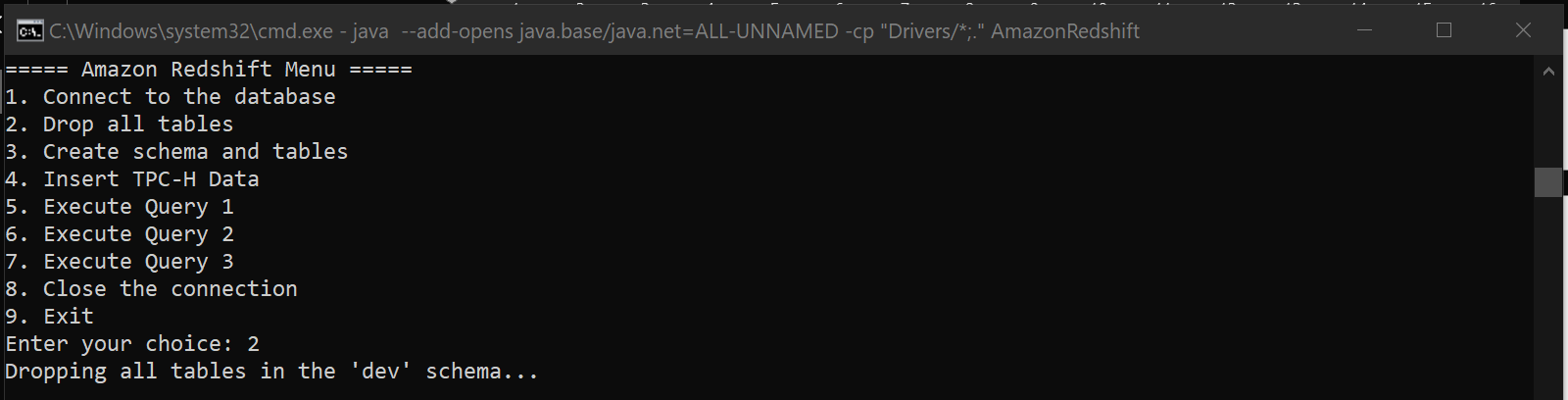
***Fig. 6.2*** *Amazon Redshift Menu displaying database operations, including options to connect, execute queries, and close the connection.*

1. Method drop() to drop all the tables from the database. Note: The database schema name will be dev. [5]

Code:

1. public void drop() {
2. System.out.println("Dropping all tables in the 'dev' schema...");
3. String dropQuery = "SELECT tablename FROM pg\_tables WHERE schemaname = 'dev'";
4. try (Statement stmt = con.createStatement()) {
5. ResultSet rs = stmt.executeQuery(dropQuery);
6. while (rs.next()) {
7. String tableName = rs.getString("tablename");
8. String dropTableQuery = "DROP TABLE IF EXISTS dev." + tableName;
9. stmt.executeUpdate(dropTableQuery);
10. System.out.println("Dropped table: " + tableName);
11. }
12. } catch (SQLException e) {
13. System.out.println("Error dropping tables: " + e.getMessage());
14. }
15. }

Output:



***Fig. 6.3*** *Amazon Redshift Menu performing the Drop all tables operation in the 'dev' schema after selecting option 2.*

1. Method create() to create the database dev and the tables. [5]

Code:

1. public void create() throws SQLException {
2. System.out.println("Creating the 'dev' schema and tables...");
3. String createSchemaQuery = "CREATE SCHEMA IF NOT EXISTS dev";
4. try (Statement stmt = con.createStatement()) {
5. stmt.executeUpdate(createSchemaQuery);
6. System.out.println("Schema 'dev' created.");
7. } catch (SQLException e) {
8. System.out.println("Error creating schema: " + e.getMessage());
9. }
10. File ddlFolder = new File("ddl");
11. File[] ddlFiles = ddlFolder.listFiles((dir, name) -> name.endsWith(".sql"));
12. if (ddlFiles != null) {
13. for (File ddlFile : ddlFiles) {
14. try {
15. String ddlQuery = new String(Files.readAllBytes(ddlFile.toPath()));
16. if (ddlFile.getName().equals("tpch\_create.sql")) {
17. // Ensure the 'tpch\_create.sql' contains CREATE TABLE queries
18. if (ddlQuery.toUpperCase().contains("CREATE TABLE")) {
19. try (Statement stmt = con.createStatement()) {
20. stmt.executeUpdate(ddlQuery);
21. System.out.println("Created table(s) from " + ddlFile.getName());
22. }
23. } else {
24. System.out.println("No CREATE TABLE queries found in tpch\_create.sql.");
25. }
26. } else {
27. System.out.println("Skipping non-CREATE TABLE SQL file: " + ddlFile.getName());
28. }
29. } catch (IOException e) {
30. System.out.println("Error reading DDL file: " + ddlFile.getName() + " - " + e.getMessage());
31. } catch (SQLException e) {
32. System.out.println("Error executing DDL query for file " + ddlFile.getName() + " - " + e.getMessage());
33. }
34. }
35. } else {
36. System.out.println("No DDL files found in 'ddl' folder.");
37. }
38. }

Output:

A screen shot of a computer

Description automatically generated

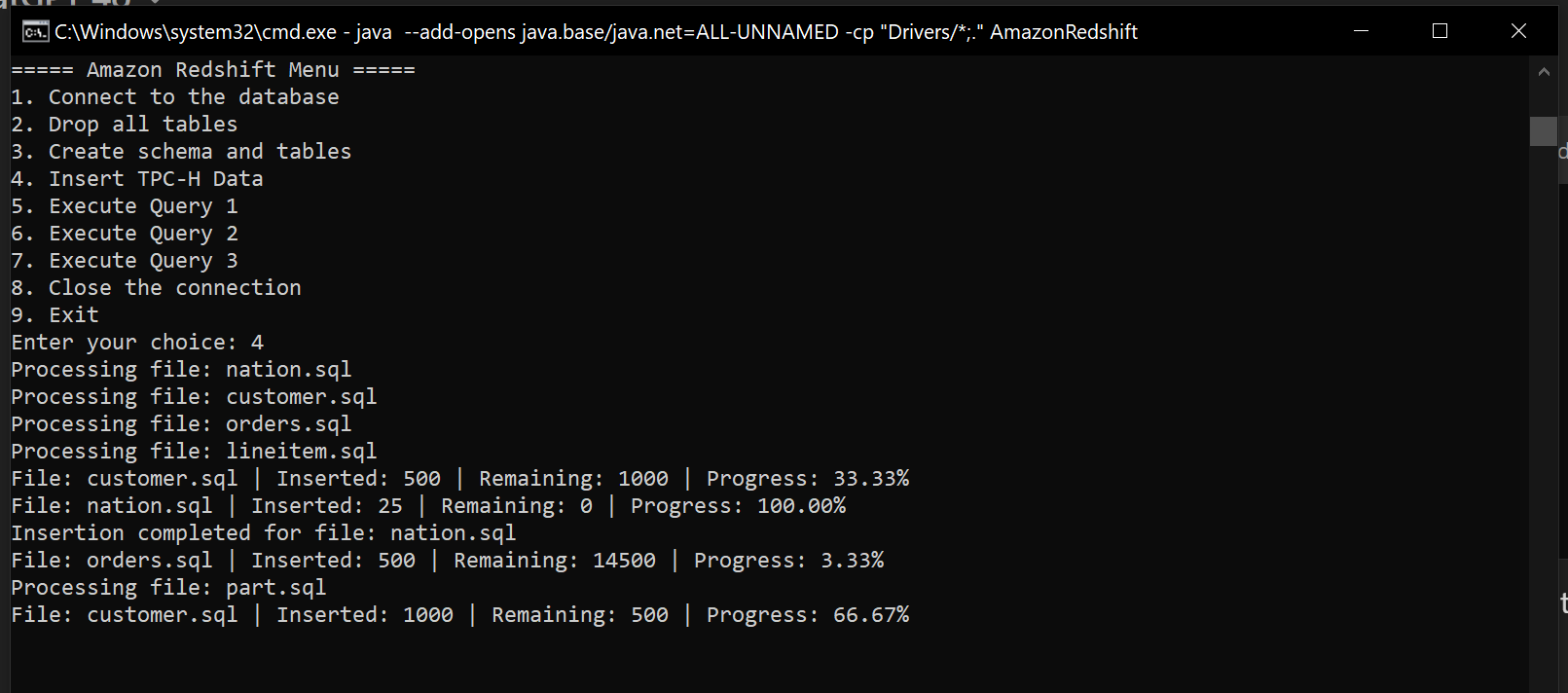
***Fig. 6.4*** *Amazon Redshift Menu executing the* ***Create schema and tables*** *operation, successfully creating the 'dev' schema and tables from SQL scripts.*

1. Write the method insert() to add the standard TPC-H data into the database. The DDL files are in the ddl folder. Hint: Files are designed so can read entire file as a string and execute it as one statement. May need to divide up into batches for large files. [10]

Code:

1. public void insert() {
2. File dataFolder = new File("ddl");
3. File[] dataFiles = dataFolder.listFiles((dir, name) -> name.endsWith(".sql") && !name.equals("tpch\_create.sql"));
4. if (dataFiles != null) {
5. // Use a CountDownLatch to wait for all insertions to complete
6. CountDownLatch latch = new CountDownLatch(dataFiles.length);
7. for (File dataFile : dataFiles) {
8. executorService.submit(() -> {
9. processFile(dataFile);
10. latch.countDown();  // Decrease latch count when each file is processed
11. });
12. }
13. try {
14. // Wait until all insert tasks are completed
15. latch.await();
16. System.out.println("All data insertions completed.");
17. } catch (InterruptedException e) {
18. System.out.println("Error waiting for insertions to complete: " + e.getMessage());
19. }
20. } else {
21. System.out.println("No data files found in the folder.");
22. }
23. }
24. private void processFile(File dataFile) {
25. try {
26. // Read the SQL query from the file
27. String sqlQuery = new String(Files.readAllBytes(dataFile.toPath()));
29. // Log the start of the process
30. System.out.println("Processing file: " + dataFile.getName());
32. // Open the Statement for executing the insert query
33. try (Statement stmt = con.createStatement()) {
34. // Split the SQL query into individual statements (assuming multiple INSERT statements in the file)
35. String[] insertStatements = sqlQuery.split(";");
36. int totalStatements = insertStatements.length; // Total number of statements in the file
38. int batchCount = 0;
39. int recordCount = 0;
41. // Process each INSERT statement
42. for (String statement : insertStatements) {
43. if (statement.trim().isEmpty()) continue;
45. // Add the statement to the batch
46. stmt.addBatch(statement.trim());
47. batchCount++;
49. // Execute the batch every 500 records
50. if (batchCount % 500 == 0) {
51. stmt.executeBatch(); // Execute the batch
52. recordCount += 500;
54. // Calculate and display progress
55. double percentageCompleted = (recordCount / (double) totalStatements) \* 100;
56. int remainingRecords = totalStatements - recordCount;
57. System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
58. dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
59. }
60. }
61. if (batchCount > 0) {
62. stmt.executeBatch();
63. recordCount += batchCount;
64. double percentageCompleted = (recordCount / (double) totalStatements) \* 100;
65. int remainingRecords = totalStatements - recordCount;
66. System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
67. dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
68. }
70. System.out.println("Insertion completed for file: " + dataFile.getName());
72. } catch (SQLException e) {
73. System.out.println("Error executing statement: " + e.getMessage());
74. e.printStackTrace();
75. }
77. } catch (IOException e) {
78. System.out.println("Error reading file: " + dataFile.getName() + " - " + e.getMessage());
79. e.printStackTrace(); // Optional: log the stack trace for more details
80. }
81. }

Output:



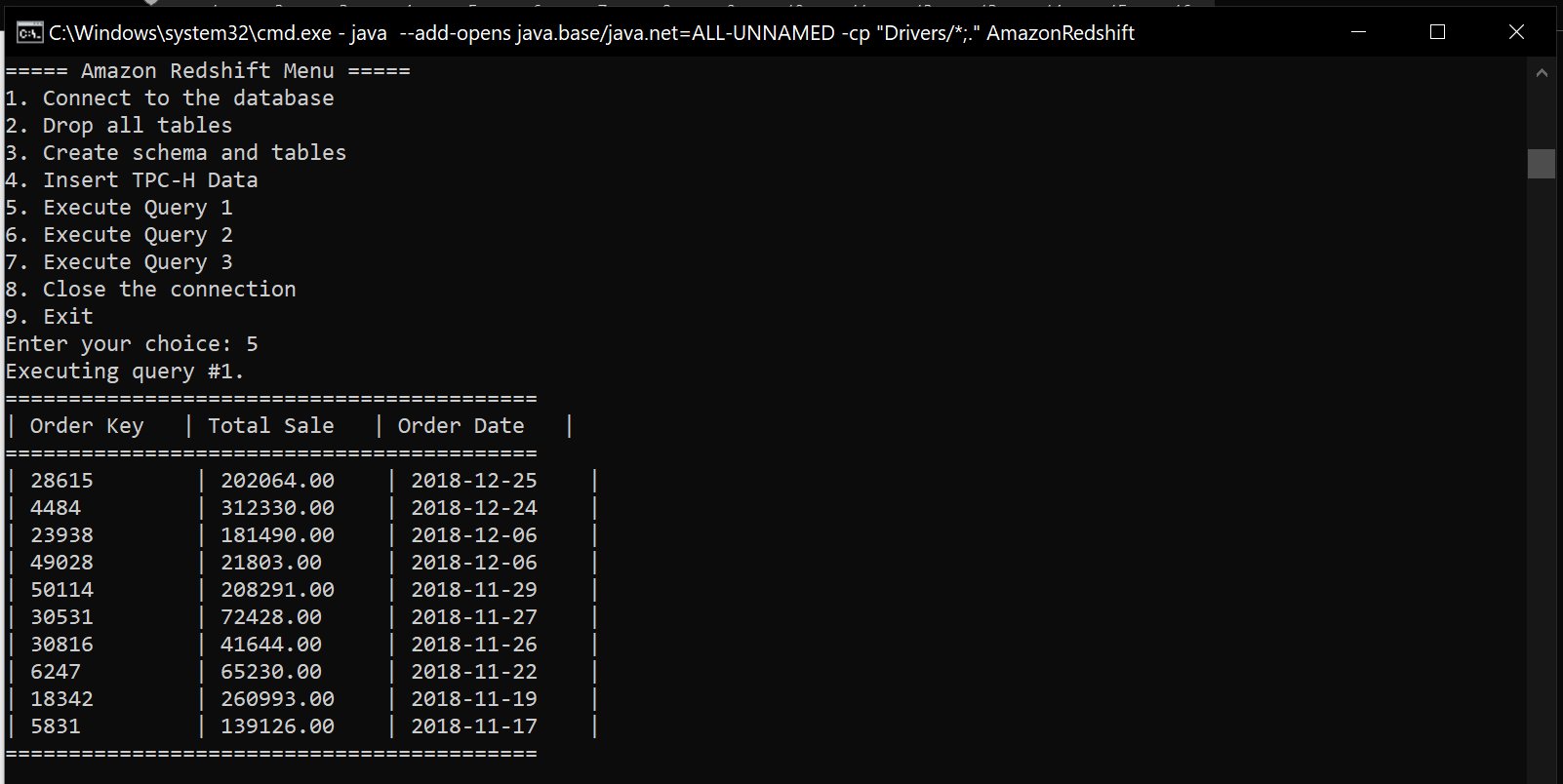
***Fig. 6.5*** *Amazon Redshift Menu executing the* ***Insert TPC-H Data*** *operation, processing multiple SQL files and showing insertion progress for each file.*

1. Write the method query1() that returns the most recent top 10 orders with the total sale and the date of the order for customers in America. [5]

Code:

1. public ResultSet query1() throws SQLException {
2. System.out.println("Executing query #1.");
3. String query = " SELECT o.O\_ORDERKEY AS order\_key, " +
4. " SUM(l.L\_EXTENDEDPRICE) AS total\_sale, " +
5. " o.O\_ORDERDATE AS order\_date " +
6. " FROM ORDERS o " +
7. " JOIN LINEITEM l ON o.O\_ORDERKEY = l.L\_ORDERKEY " +
8. " JOIN CUSTOMER c ON o.O\_CUSTKEY = c.C\_CUSTKEY " +
9. " JOIN NATION n ON c.C\_NATIONKEY = n.N\_NATIONKEY " +
10. " WHERE n.N\_NAME = 'UNITED STATES' " +
11. " GROUP BY o.O\_ORDERKEY, o.O\_ORDERDATE " +
12. " ORDER BY o.O\_ORDERDATE DESC " +
13. " LIMIT 10 ";
14. if (con == null) {
15. throw new SQLException("Connection is null. Please connect to the database first.");
16. }
18. Statement stmt = con.createStatement();
19. return stmt.executeQuery(query);
20. }

Output:



***Fig. 6.6*** *Amazon Redshift Menu executing* ***Query 1****, displaying a table with* ***Order Key****,* ***Total Sale****, and* ***Order Date*** *columns as query results.*

1. Write the method query2() that returns the customer key and the total price a customer spent in descending order, for all urgent orders that are not failed for all customers who are outside Europe and belong to the largest market segment. The largest market segment is the market segment with the most customers. [10]

Code:

1. public ResultSet query2() throws SQLException {
2. System.out.println("Executing query #2.");
3. String segmentQuery = "SELECT C\_MKTSEGMENT " +
4. "FROM CUSTOMER " +
5. "GROUP BY C\_MKTSEGMENT " +
6. "ORDER BY COUNT(C\_CUSTKEY) DESC " +
7. "LIMIT 1";
9. Statement stmt = con.createStatement();
10. ResultSet segmentResult = stmt.executeQuery(segmentQuery);
11. String largestSegment = null;
12. if (segmentResult.next()) {
13. largestSegment = segmentResult.getString("C\_MKTSEGMENT");
14. }
15. if (largestSegment != null) {
16. String query = "WITH NonEuropeanCustomers AS ( " +
17. "    SELECT C.C\_CUSTKEY " +
18. "    FROM CUSTOMER C " +
19. "    JOIN NATION N ON C.C\_NATIONKEY = N.N\_NATIONKEY " +
20. "    JOIN REGION R ON N.N\_REGIONKEY = R.R\_REGIONKEY " +
21. "    WHERE R.R\_NAME != 'EUROPE' " +
22. "), " +
23. "FilteredCustomers AS ( " +
24. "    SELECT C.C\_CUSTKEY " +
25. "    FROM CUSTOMER C " +
26. "    WHERE C.C\_MKTSEGMENT = ? " +
27. "    AND C.C\_CUSTKEY IN (SELECT C\_CUSTKEY FROM NonEuropeanCustomers) " +
28. "), " +
29. "UrgentOrders AS ( " +
30. "    SELECT O.O\_CUSTKEY AS CustomerKey, SUM(L.L\_EXTENDEDPRICE) AS TotalSpent " +
31. "    FROM ORDERS O " +
32. "    JOIN LINEITEM L ON O.O\_ORDERKEY = L.L\_ORDERKEY " +
33. "    WHERE O.O\_ORDERPRIORITY = '1-URGENT' " +
34. "      AND O.O\_ORDERSTATUS != 'F' " +
35. "      AND O.O\_CUSTKEY IN (SELECT C\_CUSTKEY FROM FilteredCustomers) " +
36. "    GROUP BY O.O\_CUSTKEY " +
37. ") " +
38. "SELECT U.CustomerKey, U.TotalSpent " +
39. "FROM UrgentOrders U " +
40. "ORDER BY U.TotalSpent DESC";
42. PreparedStatement preparedStatement = con.prepareStatement(query);
43. preparedStatement.setString(1, largestSegment); // Set the largest market segment dynamically
44. return preparedStatement.executeQuery();
45. }
46. System.out.println("No largest market segment found.");
47. return null;
48. }

Output:

===== Amazon Redshift Menu =====

1. Connect to the database

2. Drop all tables

3. Create schema and tables

4. Insert TPC-H Data

5. Execute Query 1

6. Execute Query 2

7. Execute Query 3

8. Close the connection

9. Exit

Enter your choice: 6

Executing query #2.

Query #2 Results:

==========================================

Customer Key Total Spent

==========================================

962 934276.00

1052 833829.00

103 772835.00

1279 740936.00

1061 734090.00

664 658041.00

1331 654336.00

1415 622849.00

334 620177.00

1316 611696.00

1334 601828.00

1144 601810.00

1345 574253.00

340 570314.00

1013 555097.00

1027 545509.00

694 541472.00

1253 538708.00

818 529872.00

1124 526399.00

229 522166.00

380 513735.00

835 511995.00

575 504550.00

1214 503437.00

1268 489049.00

188 475074.00

767 457100.00

995 456724.00

932 453079.00

134 438663.00

1486 437668.00

1075 425282.00

512 422890.00

1 421274.00

329 413687.00

662 404588.00

844 403041.00

649 402030.00

674 400317.00

508 395928.00

1223 386810.00

814 382438.00

1046 374573.00

803 368071.00

592 366214.00

938 365671.00

185 363520.00

580 359714.00

1414 355840.00

709 355246.00

968 353066.00

553 344943.00

298 334711.00

1163 334638.00

1100 327751.00

1009 327475.00

1115 319143.00

805 316412.00

568 312407.00

1433 308212.00

1237 308206.00

1202 308009.00

1091 306579.00

1400 305537.00

77 304452.00

1295 299133.00

860 297132.00

1430 296181.00

1235 292461.00

1453 292215.00

220 290881.00

610 284676.00

152 283341.00

476 279524.00

722 275900.00

986 271477.00

73 259207.00

428 257247.00

116 248579.00

1318 244310.00

223 244030.00

1208 243368.00

211 242005.00

1405 238524.00

1475 237839.00

944 237457.00

40 235070.00

475 234605.00

790 233068.00

280 232699.00

1201 230542.00

392 224661.00

523 224453.00

1183 220834.00

1460 217412.00

98 215161.00

1312 212232.00

653 210170.00

400 204820.00

1040 198800.00

1114 194242.00

1006 193509.00

224 191248.00

278 191091.00

1085 190796.00

1180 189196.00

1412 187654.00

865 183851.00

766 182136.00

763 175047.00

170 171616.00

1196 169535.00

1447 167667.00

1184 166621.00

64 164606.00

350 162400.00

200 159039.00

784 146507.00

113 146103.00

826 144242.00

670 143332.00

515 142196.00

448 141324.00

328 139529.00

802 138073.00

1396 130749.00

811 128303.00

347 117633.00

1358 117503.00

905 116995.00

548 115519.00

1330 112946.00

1357 109970.00

1370 108208.00

1261 106892.00

562 105994.00

904 101642.00

1385 101477.00

296 100987.00

542 99021.00

1082 96900.00

728 96476.00

793 93905.00

647 91988.00

535 91529.00

859 90163.00

518 89866.00

602 89745.00

13 86102.00

419 81138.00

121 77998.00

1277 76457.00

109 74685.00

557 73613.00

205 71835.00

32 64622.00

1468 58353.00

623 53127.00

785 48653.00

890 43337.00

1033 41460.00

386 33790.00

221 32520.00

1292 28556.00

478 10756.00

***Fig. 6.7*** *Amazon Redshift Menu executing* ***Query 2****, displaying the query results with* ***Customer Key*** *and corresponding* ***Total Spent*** *values in descending order.*

1. Write the method query3() that returns a count of all the line items that were ordered within the six years starting on April 1st, 1997 group by order priority. Make sure to sort by order priority in ascending order. [10]

Code:

1. public ResultSet query3() throws SQLException {
2. System.out.println("Executing query #3.");
3. String query = "SELECT o.O\_ORDERPRIORITY, COUNT(l.L\_LINENUMBER) AS lineitem\_count " +
4. "FROM orders o " +
5. "JOIN lineitem l ON o.O\_ORDERKEY = l.L\_ORDERKEY " +
6. "WHERE o.O\_ORDERDATE >= '1997-04-01' " +
7. "AND o.O\_ORDERDATE < DATEADD(year, 6, '1997-04-01') " +
8. "GROUP BY o.O\_ORDERPRIORITY " +
9. "ORDER BY o.O\_ORDERPRIORITY ASC";
10. Statement stmt = con.createStatement();
11. ResultSet rs = stmt.executeQuery(query);
13. return rs;
14. }

Output:



***Fig. 6.8*** *Amazon Redshift Menu executing* ***Query 3****, displaying the results with* ***Order Priority*** *and corresponding* ***Line Item Count*** *values.*

Full Code:

1. import java.math.BigDecimal;
2. import java.nio.file.Files;
3. import java.sql.Connection;
4. import java.sql.DriverManager;
5. import java.sql.PreparedStatement;
6. import java.sql.ResultSet;
7. import java.sql.ResultSetMetaData;
8. import java.sql.SQLException;
9. import java.sql.Statement;
10. import java.util.Arrays;
11. import java.util.List;
12. import java.util.Properties;
13. import java.util.Scanner;
14. import java.io.File;
15. import java.io.IOException;
16. import java.util.concurrent.CountDownLatch;
17. import java.util.concurrent.ExecutorService;
18. import java.util.concurrent.Executors;
19. /\*\*
20. \* Performs SQL DDL and SELECT queries on a MySQL database hosted on AWS RDS.
21. \*
22. \* java --add-opens java.base/java.net=ALL-UNNAMED -cp "Drivers/\*;." AmazonRedshift
23. \*/
24. public class AmazonRedshift {
25. /\*\*
26. \* Connection to database
27. \*/
28. static final String redshiftUrl = "jdbc:redshift://g23ai2087-cluster.cndskybccs1r.us-east-1.redshift.amazonaws.com:5439/dev";
29. static final String masterUsername = "admin"; // Replace with your Redshift admin username
30. static final String password = "IITj1234"; // Replace with your Redshift password
31. private static ExecutorService executorService;
32. /\*\*
33. \* Main method is only used for convenience. Use JUnit test file to verify your
34. \* answer.
35. \*
36. \* @param args
37. \*             none expected
38. \* @throws SQLException
39. \*                      if a database error occurs
40. \*/
41. public static void main(String[] args) {
42. Scanner scanner = new Scanner(System.in);
43. AmazonRedshift q = new AmazonRedshift();
44. // Initialize the executor service for parallel execution
45. executorService = Executors.newFixedThreadPool(4);
46. ResultSet rs;
47. while (true) {
48. // Display menu
49. System.out.println("\n===== Amazon Redshift Menu =====");
50. System.out.println("1. Connect to the database");
51. System.out.println("2. Drop all tables");
52. System.out.println("3. Create schema and tables");
53. System.out.println("4. Insert TPC-H Data");
54. System.out.println("5. Execute Query 1");
55. System.out.println("6. Execute Query 2");
56. System.out.println("7. Execute Query 3");
57. System.out.println("8. Close the connection");
58. System.out.println("9. Exit");
59. System.out.print("Enter your choice: ");
60. int choice = scanner.nextInt();
61. try {
62. switch (choice) {
63. case 1:
64. q.connect();
65. break;
66. case 2:
67. q.drop();
68. break;
69. case 3:
70. q.create();
71. break;
72. case 4:
73. q.insert();
74. break;
76. case 5:
77. if (q.con == null) {
78. System.out.println("Please connect to the database first.");
79. } else {
80. try {
81. rs = q.query1();
83. // Display header for the table
84. System.out.println("==========================================");
85. System.out.println("| Order Key   | Total Sale   | Order Date   |");
86. System.out.println("==========================================");
88. // Process and display each row in the ResultSet
89. while (rs.next()) {
90. System.out.printf("| %-12d | %-12.2f | %-13s |\n",
91. rs.getInt("order\_key"),        // Column alias from the query
92. rs.getDouble("total\_sale"),   // Column alias from the query
93. rs.getDate("order\_date"));    // Column alias from the query
94. }
96. System.out.println("==========================================");
97. } catch (SQLException e) {
98. System.out.println("Error processing result set: " + e.getMessage());
99. }
100. }
101. break;
102. case 6:
103. try {
104. rs = q.query2();
105. if (rs != null) {
106. System.out.println("Query #2 Results:");
107. System.out.println("==========================================");
108. System.out.printf("%-15s %-20s%n", "Customer Key", "Total Spent");
109. System.out.println("==========================================");
110. while (rs.next()) {
111. System.out.printf("%-15d %-20.2f%n",
112. rs.getInt("CustomerKey"),
113. rs.getDouble("TotalSpent"));
114. }
115. System.out.println("==========================================");
116. } else {
117. System.out.println("No results found for Query #2.");
118. }
119. } catch (SQLException e) {
120. System.out.println("Error executing Query #2: " + e.getMessage());
121. }
122. break;
123. case 7:
124. if (q.con == null) {
125. System.out.println("Please connect to the database first.");
126. } else {
127. rs = q.query3();
128. System.out.println("==========================================");
129. System.out.println("| Order Priority   | Line Item Count     |");
130. System.out.println("==========================================");
131. while (rs.next()) {
132. String orderPriority = rs.getString("O\_ORDERPRIORITY");
133. int lineItemCount = rs.getInt("lineitem\_count");
134. System.out.printf("| %-16s | %-18d |\n", orderPriority, lineItemCount);
135. }
136. System.out.println("==========================================");
137. }
138. break;
139. case 8:
140. q.close();
141. break;
142. case 9:
143. System.out.println("Exiting program...");
144. scanner.close();
145. System.exit(0);  // Exit the program
146. break;
147. default:
148. System.out.println("Invalid choice. Please try again.");
149. break;
150. }
151. } catch (SQLException e) {
152. System.out.println("Error: " + e.getMessage());
153. } catch (Exception e) {
154. System.out.println("An unexpected error occurred: " + e.getMessage());
155. }
156. }
157. }
158. /\*\*
159. \* Makes a connection to the database and returns connection to caller.
160. \*
161. \* @return
162. \*         connection
163. \* @throws SQLException
164. \*                      if an error occurs
165. \*/
167. // Redshift connection details

170. private Connection con;
171. public Connection connect() {
172. try {
173. Class.forName("com.amazon.redshift.jdbc42.Driver");
174. Properties properties = new Properties();
175. properties.setProperty("user", masterUsername);
176. properties.setProperty("password", password);
177. this.con = DriverManager.getConnection(redshiftUrl, properties);
178. System.out.println("Connection to Redshift established successfully.");
179. } catch (ClassNotFoundException e) {
180. System.out.println("Error: Redshift JDBC driver not found.");
181. e.printStackTrace();
182. } catch (SQLException e) {
183. System.out.println("Failed to connect to the database.");
184. e.printStackTrace();
185. }
186. return con;
187. }
188. /\*\*
189. \* Closes connection to database.
190. \*/
191. public void close() {
192. System.out.println("Closing database connection.");
193. try {
194. if (con != null && !con.isClosed()) {
195. con.close();
196. }
197. } catch (SQLException e) {
198. System.out.println("Error closing the connection.");
199. }
200. }
201. public void drop() {
202. System.out.println("Dropping all tables in the 'dev' schema...");
203. String dropQuery = "SELECT tablename FROM pg\_tables WHERE schemaname = 'dev'";
204. try (Statement stmt = con.createStatement()) {
205. ResultSet rs = stmt.executeQuery(dropQuery);
206. while (rs.next()) {
207. String tableName = rs.getString("tablename");
208. String dropTableQuery = "DROP TABLE IF EXISTS dev." + tableName;
209. stmt.executeUpdate(dropTableQuery);
210. System.out.println("Dropped table: " + tableName);
211. }
212. } catch (SQLException e) {
213. System.out.println("Error dropping tables: " + e.getMessage());
214. }
215. }
216. public void create() throws SQLException {
217. System.out.println("Creating the 'dev' schema and tables...");
218. String createSchemaQuery = "CREATE SCHEMA IF NOT EXISTS dev";
219. try (Statement stmt = con.createStatement()) {
220. stmt.executeUpdate(createSchemaQuery);
221. System.out.println("Schema 'dev' created.");
222. } catch (SQLException e) {
223. System.out.println("Error creating schema: " + e.getMessage());
224. }
225. File ddlFolder = new File("ddl");
226. File[] ddlFiles = ddlFolder.listFiles((dir, name) -> name.endsWith(".sql"));
227. if (ddlFiles != null) {
228. for (File ddlFile : ddlFiles) {
229. try {
230. String ddlQuery = new String(Files.readAllBytes(ddlFile.toPath()));
231. if (ddlFile.getName().equals("tpch\_create.sql")) {
232. // Ensure the 'tpch\_create.sql' contains CREATE TABLE queries
233. if (ddlQuery.toUpperCase().contains("CREATE TABLE")) {
234. try (Statement stmt = con.createStatement()) {
235. stmt.executeUpdate(ddlQuery);
236. System.out.println("Created table(s) from " + ddlFile.getName());
237. }
238. } else {
239. System.out.println("No CREATE TABLE queries found in tpch\_create.sql.");
240. }
241. } else {
242. System.out.println("Skipping non-CREATE TABLE SQL file: " + ddlFile.getName());
243. }
244. } catch (IOException e) {
245. System.out.println("Error reading DDL file: " + ddlFile.getName() + " - " + e.getMessage());
246. } catch (SQLException e) {
247. System.out.println("Error executing DDL query for file " + ddlFile.getName() + " - " + e.getMessage());
248. }
249. }
250. } else {
251. System.out.println("No DDL files found in 'ddl' folder.");
252. }
253. }
254. public void insert() {
255. File dataFolder = new File("ddl");
256. File[] dataFiles = dataFolder.listFiles((dir, name) -> name.endsWith(".sql") && !name.equals("tpch\_create.sql"));
257. if (dataFiles != null) {
258. // Use a CountDownLatch to wait for all insertions to complete
259. CountDownLatch latch = new CountDownLatch(dataFiles.length);
260. for (File dataFile : dataFiles) {
261. executorService.submit(() -> {
262. processFile(dataFile);
263. latch.countDown();  // Decrease latch count when each file is processed
264. });
265. }
266. try {
267. // Wait until all insert tasks are completed
268. latch.await();
269. System.out.println("All data insertions completed.");
270. } catch (InterruptedException e) {
271. System.out.println("Error waiting for insertions to complete: " + e.getMessage());
272. }
273. } else {
274. System.out.println("No data files found in the folder.");
275. }
276. }
277. private void processFile(File dataFile) {
278. try {
279. // Read the SQL query from the file
280. String sqlQuery = new String(Files.readAllBytes(dataFile.toPath()));
282. // Log the start of the process
283. System.out.println("Processing file: " + dataFile.getName());
285. // Open the Statement for executing the insert query
286. try (Statement stmt = con.createStatement()) {
287. // Split the SQL query into individual statements (assuming multiple INSERT statements in the file)
288. String[] insertStatements = sqlQuery.split(";");
289. int totalStatements = insertStatements.length; // Total number of statements in the file
291. int batchCount = 0;
292. int recordCount = 0;
294. // Process each INSERT statement
295. for (String statement : insertStatements) {
296. if (statement.trim().isEmpty()) continue;
298. // Add the statement to the batch
299. stmt.addBatch(statement.trim());
300. batchCount++;
302. // Execute the batch every 500 records
303. if (batchCount % 500 == 0) {
304. stmt.executeBatch(); // Execute the batch
305. recordCount += 500;
307. // Calculate and display progress
308. double percentageCompleted = (recordCount / (double) totalStatements) \* 100;
309. int remainingRecords = totalStatements - recordCount;
310. System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
311. dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
312. }
313. }
314. if (batchCount > 0) {
315. stmt.executeBatch();
316. recordCount += batchCount;
317. double percentageCompleted = (recordCount / (double) totalStatements) \* 100;
318. int remainingRecords = totalStatements - recordCount;
319. System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
320. dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
321. }
323. System.out.println("Insertion completed for file: " + dataFile.getName());
325. } catch (SQLException e) {
326. System.out.println("Error executing statement: " + e.getMessage());
327. e.printStackTrace();
328. }
330. } catch (IOException e) {
331. System.out.println("Error reading file: " + dataFile.getName() + " - " + e.getMessage());
332. e.printStackTrace(); // Optional: log the stack trace for more details
333. }
334. }
336. /\*\*
337. \* Query returns the most recent top 10 orders with the total sale and the date
338. \* of the
339. \* order in `America`.
340. \*
341. \* @return
342. \*         ResultSet
343. \* @throws SQLException
344. \*                      if an error occurs
345. \*/
346. public ResultSet query1() throws SQLException {
347. System.out.println("Executing query #1.");
348. String query = " SELECT o.O\_ORDERKEY AS order\_key, " +
349. " SUM(l.L\_EXTENDEDPRICE) AS total\_sale, " +
350. " o.O\_ORDERDATE AS order\_date " +
351. " FROM ORDERS o " +
352. " JOIN LINEITEM l ON o.O\_ORDERKEY = l.L\_ORDERKEY " +
353. " JOIN CUSTOMER c ON o.O\_CUSTKEY = c.C\_CUSTKEY " +
354. " JOIN NATION n ON c.C\_NATIONKEY = n.N\_NATIONKEY " +
355. " WHERE n.N\_NAME = 'UNITED STATES' " +
356. " GROUP BY o.O\_ORDERKEY, o.O\_ORDERDATE " +
357. " ORDER BY o.O\_ORDERDATE DESC " +
358. " LIMIT 10 ";
359. if (con == null) {
360. throw new SQLException("Connection is null. Please connect to the database first.");
361. }
363. Statement stmt = con.createStatement();
364. return stmt.executeQuery(query);
365. }

368. /\*\*
369. \* Query returns the customer key and the total price a customer spent in
370. \* descending
371. \* order, for all urgent orders that are not failed for all customers who are
372. \* outside Europe belonging
373. \* to the highest market segment.
374. \*
375. \* @return
376. \*         ResultSet
377. \* @throws SQLException
378. \*                      if an error occurs
379. \*/
380. public ResultSet query2() throws SQLException {
381. System.out.println("Executing query #2.");
382. String segmentQuery = "SELECT C\_MKTSEGMENT " +
383. "FROM CUSTOMER " +
384. "GROUP BY C\_MKTSEGMENT " +
385. "ORDER BY COUNT(C\_CUSTKEY) DESC " +
386. "LIMIT 1";
388. Statement stmt = con.createStatement();
389. ResultSet segmentResult = stmt.executeQuery(segmentQuery);
390. String largestSegment = null;
391. if (segmentResult.next()) {
392. largestSegment = segmentResult.getString("C\_MKTSEGMENT");
393. }
394. if (largestSegment != null) {
395. String query = "WITH NonEuropeanCustomers AS ( " +
396. "    SELECT C.C\_CUSTKEY " +
397. "    FROM CUSTOMER C " +
398. "    JOIN NATION N ON C.C\_NATIONKEY = N.N\_NATIONKEY " +
399. "    JOIN REGION R ON N.N\_REGIONKEY = R.R\_REGIONKEY " +
400. "    WHERE R.R\_NAME != 'EUROPE' " +
401. "), " +
402. "FilteredCustomers AS ( " +
403. "    SELECT C.C\_CUSTKEY " +
404. "    FROM CUSTOMER C " +
405. "    WHERE C.C\_MKTSEGMENT = ? " +
406. "    AND C.C\_CUSTKEY IN (SELECT C\_CUSTKEY FROM NonEuropeanCustomers) " +
407. "), " +
408. "UrgentOrders AS ( " +
409. "    SELECT O.O\_CUSTKEY AS CustomerKey, SUM(L.L\_EXTENDEDPRICE) AS TotalSpent " +
410. "    FROM ORDERS O " +
411. "    JOIN LINEITEM L ON O.O\_ORDERKEY = L.L\_ORDERKEY " +
412. "    WHERE O.O\_ORDERPRIORITY = '1-URGENT' " +
413. "      AND O.O\_ORDERSTATUS != 'F' " +
414. "      AND O.O\_CUSTKEY IN (SELECT C\_CUSTKEY FROM FilteredCustomers) " +
415. "    GROUP BY O.O\_CUSTKEY " +
416. ") " +
417. "SELECT U.CustomerKey, U.TotalSpent " +
418. "FROM UrgentOrders U " +
419. "ORDER BY U.TotalSpent DESC";
421. PreparedStatement preparedStatement = con.prepareStatement(query);
422. preparedStatement.setString(1, largestSegment); // Set the largest market segment dynamically
423. return preparedStatement.executeQuery();
424. }
425. System.out.println("No largest market segment found.");
426. return null;
427. }
429. /\*\*
430. \* Query returns all the lineitems that was ordered within the six years from
431. \* January 4th,
432. \* 1997 and the orderpriority in ascending order.
433. \*
434. \* @return
435. \*         ResultSet
436. \* @throws SQLException
437. \*                      if an error occurs
438. \*/
439. public ResultSet query3() throws SQLException {
440. System.out.println("Executing query #3.");
441. String query = "SELECT o.O\_ORDERPRIORITY, COUNT(l.L\_LINENUMBER) AS lineitem\_count " +
442. "FROM orders o " +
443. "JOIN lineitem l ON o.O\_ORDERKEY = l.L\_ORDERKEY " +
444. "WHERE o.O\_ORDERDATE >= '1997-04-01' " +
445. "AND o.O\_ORDERDATE < DATEADD(year, 6, '1997-04-01') " +
446. "GROUP BY o.O\_ORDERPRIORITY " +
447. "ORDER BY o.O\_ORDERPRIORITY ASC";
448. Statement stmt = con.createStatement();
449. ResultSet rs = stmt.executeQuery(query);
451. return rs;
452. }
454. /\*
455. \* Do not change anything below here.
456. \*/
457. /\*\*
458. \* Converts a ResultSet to a string with a given number of rows displayed.
459. \* Total rows are determined but only the first few are put into a string.
460. \*
461. \* @param rst
462. \*                ResultSet
463. \* @param maxrows
464. \*                maximum number of rows to display
465. \* @return
466. \*         String form of results
467. \* @throws SQLException
468. \*                      if a database error occurs
469. \*/
470. public static String resultSetToString(ResultSet rst, int maxrows) throws SQLException {
471. StringBuffer buf = new StringBuffer(5000);
472. int rowCount = 0;
473. ResultSetMetaData meta = rst.getMetaData();
474. buf.append("Total columns: " + meta.getColumnCount());
475. buf.append('\n');
476. if (meta.getColumnCount() > 0)
477. buf.append(meta.getColumnName(1));
478. for (int j = 2; j <= meta.getColumnCount(); j++)
479. buf.append(", " + meta.getColumnName(j));
480. buf.append('\n');
481. while (rst.next()) {
482. if (rowCount < maxrows) {
483. for (int j = 0; j < meta.getColumnCount(); j++) {
484. Object obj = rst.getObject(j + 1);
485. buf.append(obj);
486. if (j != meta.getColumnCount() - 1)
487. buf.append(", ");
488. }
489. buf.append('\n');
490. }
491. rowCount++;
492. }
493. buf.append("Total results: " + rowCount);
494. return buf.toString();
495. }
496. /\*\*
497. \* Converts ResultSetMetaData into a string.
498. \*
499. \* @param meta
500. \*             ResultSetMetaData
501. \* @return
502. \*         string form of metadata
503. \* @throws SQLException
504. \*                      if a database error occurs
505. \*/
506. public static String resultSetMetaDataToString(ResultSetMetaData meta) throws SQLException {
507. StringBuffer buf = new StringBuffer(5000);
508. buf.append(meta.getColumnName(1) + " (" + meta.getColumnLabel(1) + ", " +
509. meta.getColumnType(1) + "-" + meta.getColumnTypeName(1) + ", " +
510. meta.getColumnDisplaySize(1) + ", " + meta.getPrecision(1) + ", " +
511. meta.getScale(1) + ")");
512. for (int j = 2; j <= meta.getColumnCount(); j++) {
513. buf.append(", " + meta.getColumnName(j) + " (" + meta.getColumnLabel(j) + ", " +
514. meta.getColumnType(j) + "-" + meta.getColumnTypeName(j) + ", " +
515. meta.getColumnDisplaySize(j) + ", " + meta.getPrecision(j) + ", " +
516. meta.getScale(j) + ")");
517. }
518. return buf.toString();
519. }
520. }